CS 267: Applications of Parallel Computers

Lecture 3:
Introduction to Parallel Architectures and Programming Models

David H. Bailey

based on notes by J. Demmel and D. Culler

http://www.nersc.gov/~dhbailey/cs267

Recap of Last Lecture

° **The actual performance of a simple program can depend in complicated ways on the architecture.**

° **Slight changes in the program may change the performance significantly.**

° **For best performance, we must take the architecture into account, even on single processor systems.**

° **Since performance is so complicated, we need simple models to help us design efficient algorithms.**

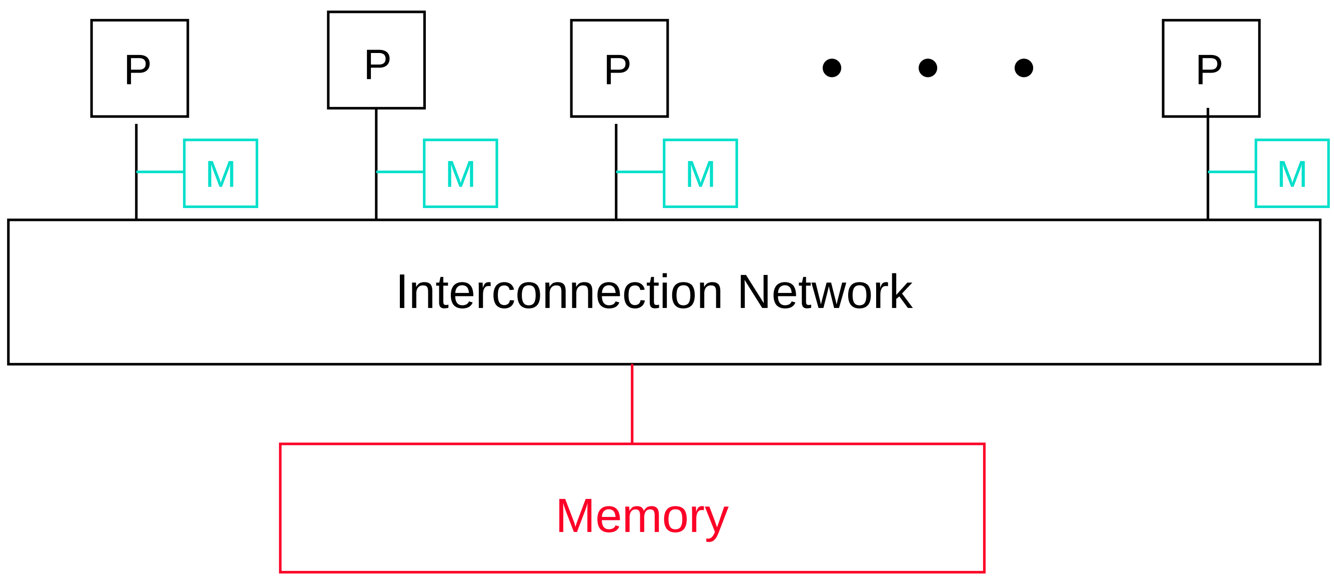
° **We illustrated with a common technique for improving cache performance, called blocking, applied to matrix multiplication.**

Outline

- Parallel machines and programming models
- Steps in writing a parallel program
- Cost modeling and performance trade-offs

Parallel Machines and Programming Models

A generic parallel architecture



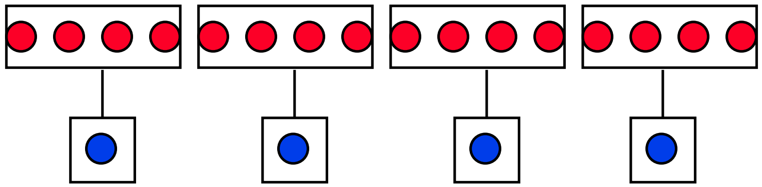
- Where is the memory physically located?

Parallel Programming Models

- **Control**
  - **How is parallelism created?**
  - **What orderings exist between operations?**
  - **How do different threads of control synchronize?**
- **Data**
  - **What data is private vs. shared?**
  - **How is logically shared data accessed or communicated?**
- **Operations**
  - **What are the atomic operations?**
- **Cost**
  - **How do we account for the cost of each of the above?**

Trivial Example

- $$\sum_{i=0}^{n-1} f(A[i])$$
- **Parallel Decomposition:**
  - **Each evaluation and each partial sum is a task.**
- **Assign n/p numbers to each of p procs**
  - **Each computes independent "private" results and partial sum.**
  - **One (or all) collects the p partial sums and computes the global sum.**

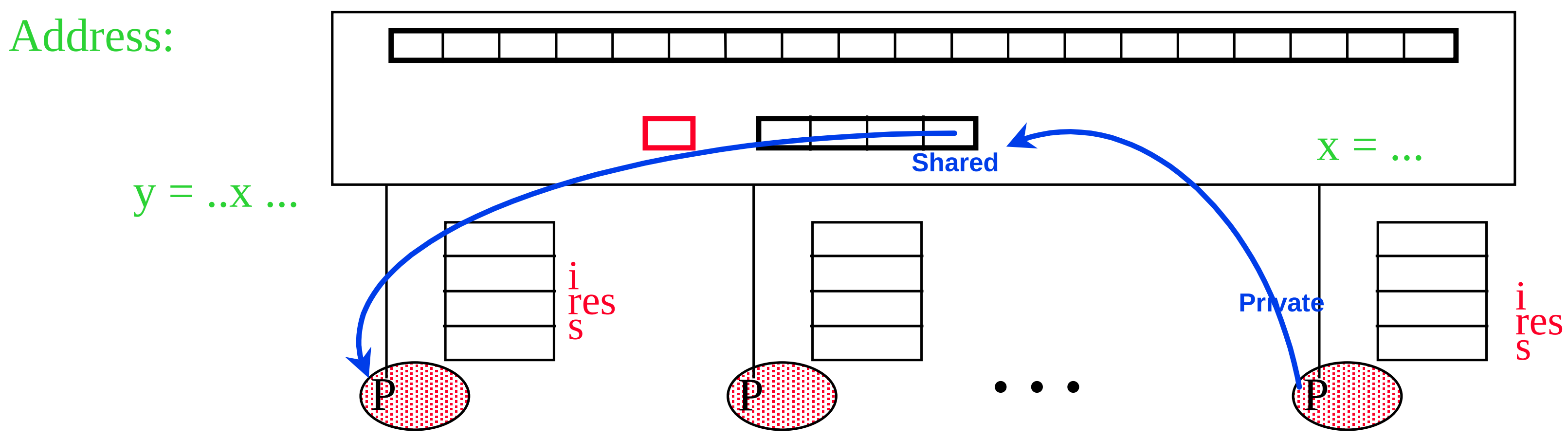


Two Classes of Data:

- **Logically Shared**
  - **The original n numbers, the global sum.**
- **Logically Private**
  - **The individual function evaluations.**
  - **What about the individual partial sums?**

Programming Model 1: Shared Address Space

- Program consists of a collection of threads of control.
- Each has a set of private variables, e.g. local variables on the stack.
- Collectively with a set of shared variables, e.g., static variables, shared common blocks, global heap.
- Threads communicate implicitly by writing and reading shared variables.
- Threads coordinate explicitly by synchronization operations on shared variables -- writing and reading flags, locks or semaphores.
- Like concurrent programming on a uniprocessor.



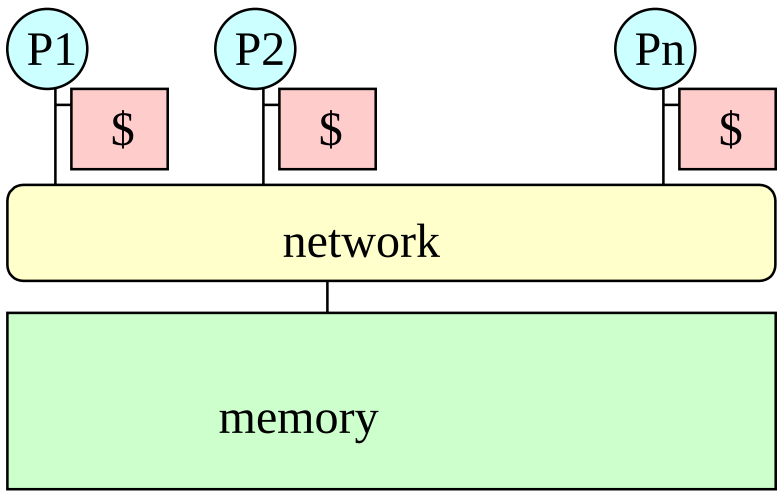
Machine Model 1: Shared Memory Multiprocessor

° Processors all connected to a large shared memory.

° "Local" memory is not (usually) part of the hardware.

  • Sun, DEC, Intel SMPs in Millennium, SGI Origin

° Cost: much cheaper to access data in cache than in main memory.



° Machine model 1a: Shared Address Space Machine (Cray T3E)

  • Replace caches by local memories (in abstract machine model).
  • This affects the cost model -- repeatedly accessed data should be copied to local memory.

Shared Memory Code for Computing a Sum

Thread 1

```
[s = 0 initially]
local_s1= 0
for i = 0, n/2-1
   local_s1 = local_s1 + f(A[i])
s = s + local_s1
```

Thread 2

```
[s = 0 initially]
local_s2 = 0
for i = n/2, n-1
   local_s2= local_s2 + f(A[i])
s = s +local_s2
```

What could go wrong?

Pitfall and Solution via Synchronization

° Pitfall in computing a global sum s = local_s1 + local_s2:

Thread 1 (initially s=0)	Thread 2 (initially s=0)
load s [from mem to reg]	
	load s [from mem to reg; initially 0]
s = s+local_s1 [=local_s1, in reg]	s = s+local_s2 [=local_s2, in reg]
store s [from reg to mem]	store s [from reg to mem]

Time

° Instructions from different threads can be interleaved arbitrarily.

° What can final result s stored in memory be?

° Problem: race condition.

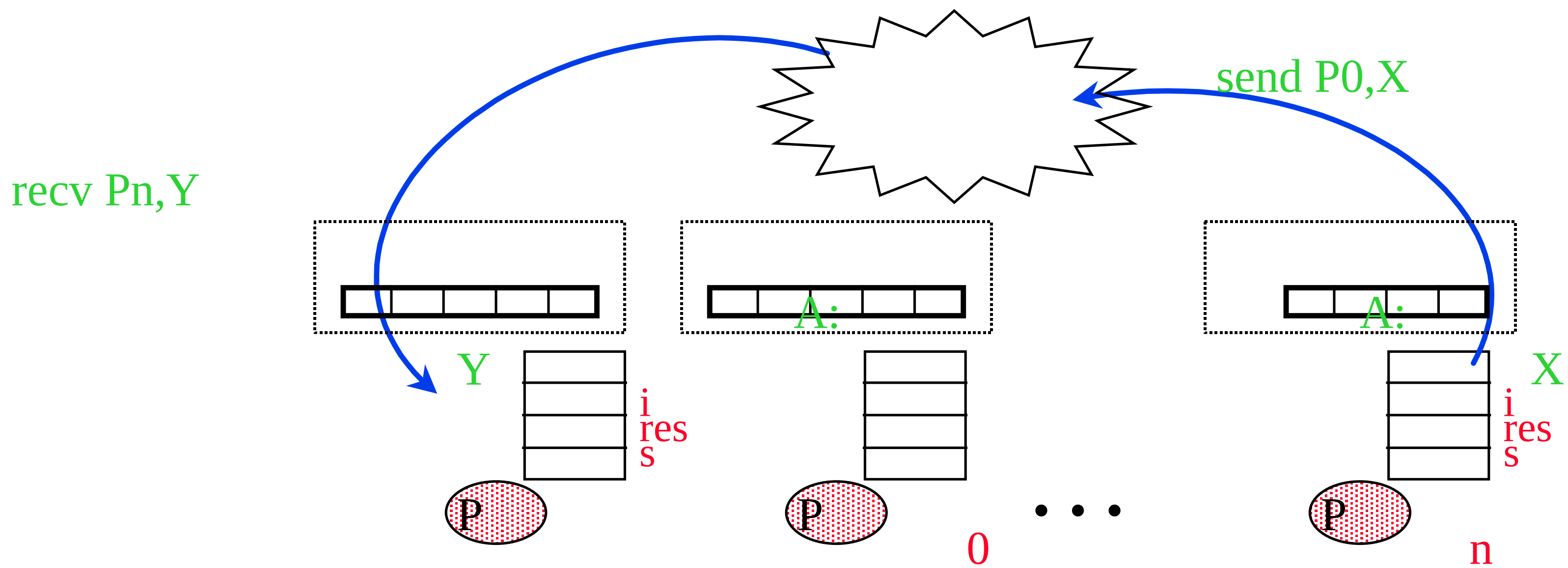
° Possible solution: mutual exclusion with locks

Thread 1	Thread 2
lock	lock
load s	load s
s = s+local_s1	s = s+local_s2
store s	store s
unlock	unlock

° Locks must be atomic (execute completely without interruption).

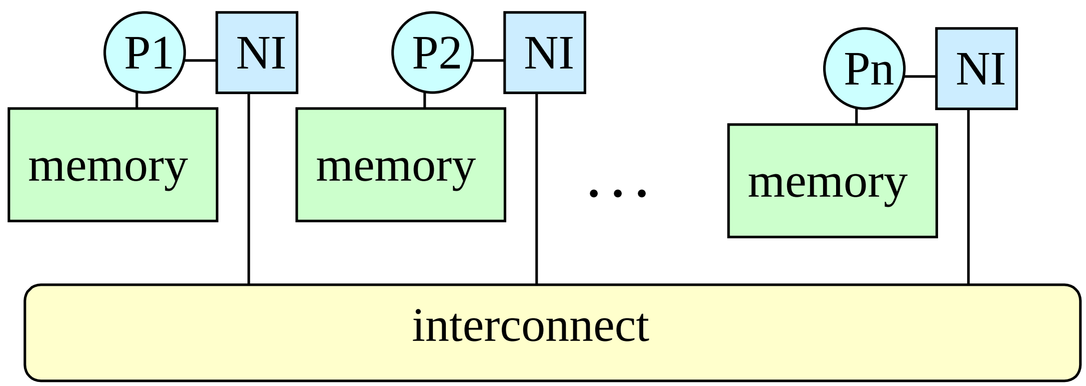
Programming Model 2: Message Passing

- Program consists of a collection of named processes.
- Thread of control plus local address space -- NO shared data.
- Local variables, static variables, common blocks, heap.
- Processes communicate by explicit data transfers -- matching send and receive pair by source and destination processors.
- Coordination is implicit in every communication event.
- Logically shared data is partitioned over local processes.
- Like distributed programming -- program with MPI, PVM.



Machine Model 2: Distributed Memory

- **Cray T3E (too!), IBM SP2, NOW, Millennium.**
- **Each processor is connected to its own memory and cache but cannot directly access another processor’s memory.**
- **Each “node” has a network interface (NI) for all communication and synchronization.**



Computing s = x(1)+x(2) on each processor

° **First possible solution:**

Processor 1
  send xlocal, proc2
    [xlocal = x(1)]
  receive xremote, proc2
  s = xlocal + xremote

Processor 2
  receive xremote, proc1
  send xlocal, proc1
    [xlocal = x(2)]
  s = xlocal + xremote

° **Second possible solution -- what could go wrong?**

Processor 1
  send xlocal, proc2
    [xlocal = x(1)]
  receive xremote, proc2
  s = xlocal + xremote

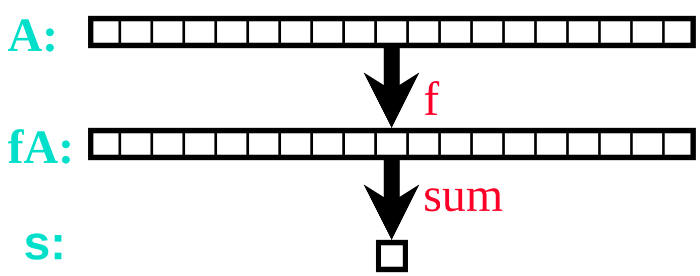
Processor 2
  send xlocal, proc1
    [xlocal = x(2)]
  receive xremote, proc1
  s = xlocal + xremote

° **What if send/receive acts like the telephone system? The post office?**

Programming Model 3: Data Parallel

- Single sequential thread of control consisting of parallel operations.
- Parallel operations applied to all (or a defined subset) of a data structure.
- Communication is implicit in parallel operators and “shifted” data structures.
- Elegant and easy to understand and reason about.
- Like marching in a regiment.
- Used by Matlab.
- Drawback: not all problems fit this model.

A = array of all data
fA = f(A)
s = sum(fA)



Machine Model 3: SIMD System

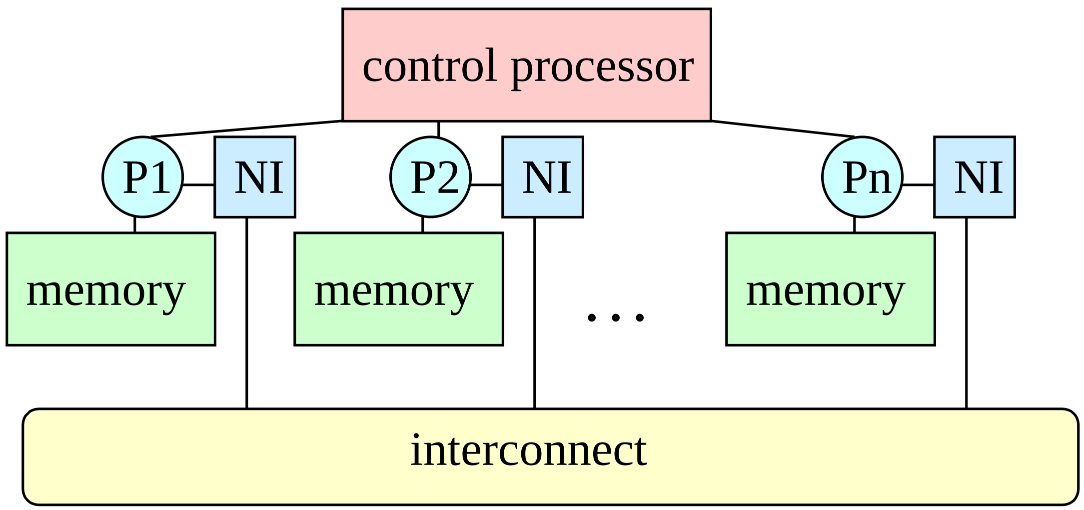
° A large number of (usually) small processors.

° A single "control processor" issues each instruction.

° Each processor executes the same instruction.

° Some processors may be turned off on some instructions.

° Machines are not popular (CM2), but programming model is.



- Implemented by mapping n-fold parallelism to p processors.
- Mostly done in the compilers (HPF = High Performance Fortran).

Machine Model 4: Clusters of SMPs

° **Since small shared memory machines (SMPs) are the fastest commodity machine, why not build a larger machine by connecting many of them with a network?**

° **CLUMP = Cluster of SMPs.**

° **Shared memory within one SMP, but message passing outside of an SMP.**

° **Millennium, ASCI Red (Intel), ...**

° **Two programming models:**

- **Treat machine as "flat", always use message passing, even within SMP (simple, but ignores an important part of memory hierarchy).**
- **Expose two layers: shared memory and message passing (usually higher performance, but ugly to program).**

Programming Model 5: Bulk Synchronous

- **Used within the message passing or shared memory models as a programming convention.**
- **Phases are separated by global barriers:**
  - **Compute phases: all operate on local data (in distributed memory) or read access to global data (in shared memory).**
  - **Communication phases: all participate in rearrangement or reduction of global data.**
- **Generally all doing the “same thing” in a phase:**
  - **all do f, but may all do different things within f.**
- **Features the simplicity of data parallelism, but without the restrictions of a strict data parallel model.**

Summary So Far

- **Historically, each parallel machine was unique, along with its programming model and programming language.**
- **It was necessary to through away software and start over with each new kind of machine - ugh.**
- **Now we distinguish the programming model from the underlying machine, so we can write portably correct codes that run on many machines.**
  - **MPI now the most portable option, but can be tedious.**
- **Writing portably fast code requires tuning for the architecture.**
  - **Algorithm design challenge is to make this process easy.**
  - **Example: picking a blocksize, not rewriting whole algorithm.**

Steps in Writing Parallel Programs

Creating a Parallel Program

- Identify work that can be done in parallel.
- Partition work and perhaps data among logical processes (threads).
- Manage the data access, communication, synchronization.
- Goal: maximize speedup due to parallelism

$$\text{Speedup}_{prob}(\text{P procs}) = \frac{\text{Time to solve prob with "best" sequential solution}}{\text{Time to solve prob in parallel on P processors}}$$

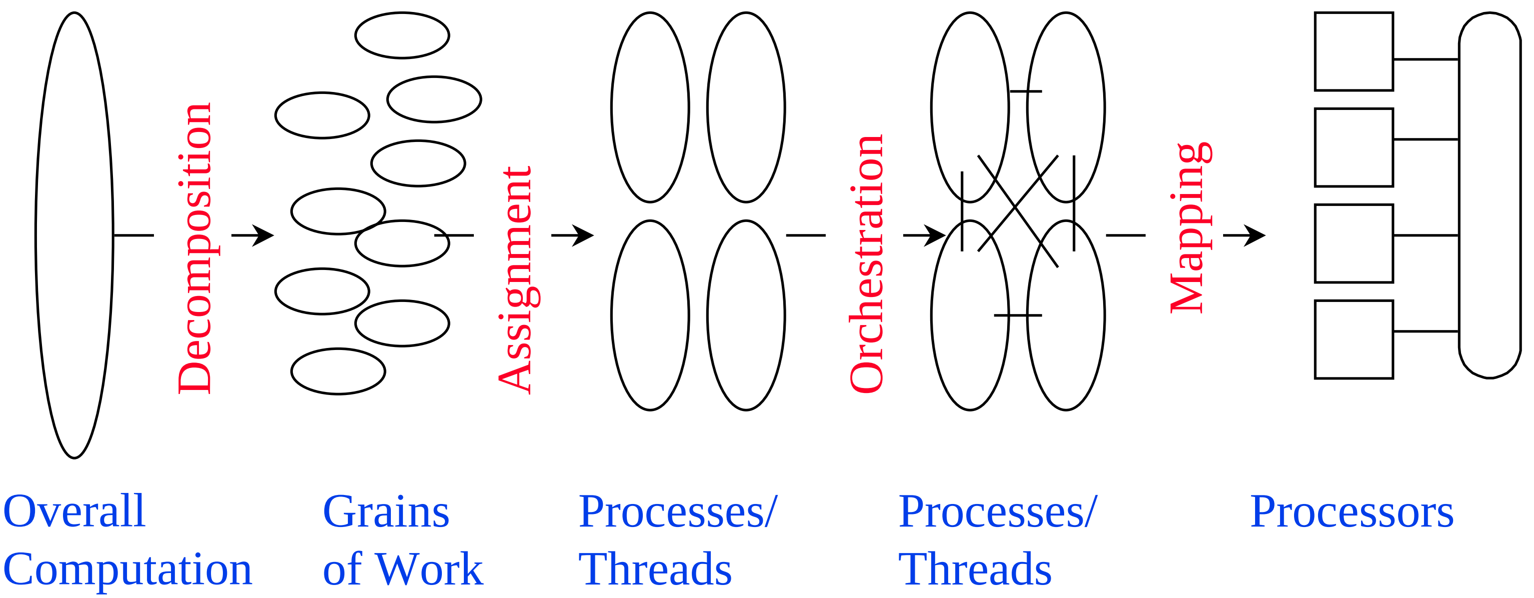
$$\le P \qquad \text{(Brent's Theorem)}$$

$$\text{Efficiency(P)} = \text{Speedup(P)} / P$$

$$\le 1$$

- Key question is when you can solve each piece:
  - statically, if information is known in advance.
  - dynamically, otherwise.

Steps in the Process



- **Task:** arbitrarily defined piece of work that forms the basic unit of concurrency.
- **Process/Thread:** abstract entity that performs tasks:
  - **tasks are assigned to threads via an assignment mechanism.**
  - **threads must coordinate to accomplish their collective tasks.**
- **Processor:** physical entity that executes a thread.

Decomposition

- Break the overall computation into individual grains of work (tasks).
  - Identify concurrency and decide at what level to exploit it.
  - Concurrency may be statically identifiable or may vary dynamically.
  - It may depend only on problem size, or it may depend on the particular input data.
- Goal: identify enough tasks to keep the target range of processors busy, but not too many.
  - Establishes upper limit on number of useful processors (i.e., scaling).
- Tradeoff: sufficient concurrency vs. task control overhead.

Assignment

- Determine mechanism to divide work among threads
  - Functional partitioning:
    - Assign logically distinct aspects of work to different thread, e.g. pipelining.
  - Structural mechanisms:
    - Assign iterations of “parallel loop” according to a simple rule, e.g. proc j gets iterates j*n/p through (j+1)*n/p-1.
    - Throw tasks in a bowl (task queue) and let threads feed.
  - Data/domain decomposition:
    - Data describing the problem has a natural decomposition.
    - Break up the data and assign work associated with regions, e.g. parts of physical system being simulated.
- Goals:
  - Balance the workload to keep everyone busy (all the time).
  - Allow efficient orchestration.

Orchestration

° **Provide a means of**

- **Naming and accessing shared data.**
- **Communication and coordination among threads of control.**

Goals:

- **Correctness of parallel solution -- respect the inherent dependencies within the algorithm.**
- **Avoid serialization.**
- **Reduce cost of communication, synchronization, and management.**
- **Preserve locality of data reference.**

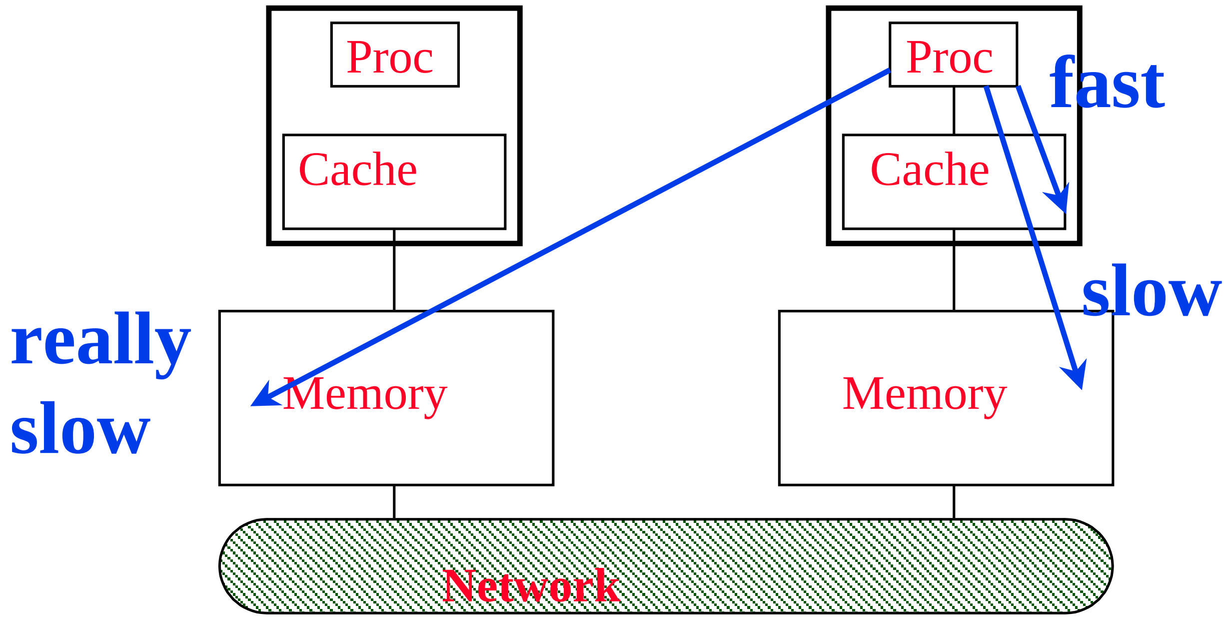
Mapping

° **Binding processes to physical processors.**

° **Time to reach processor across network does not depend on which processor (roughly).**

- **lots of old literature on “network topology”, no longer so important.**

° **Basic issue is how many remote accesses.**



Example

- s = f(A[1]) + … + f(A[n])
- Decomposition
  - computing each f(A[j])
  - n-fold parallelism, where n may be >> p
  - computing sum s
- Assignment
  - thread k sums sk = f(A[k*n/p]) + … + f(A[(k+1)*n/p-1])
  - thread 1 sums s = s1+ … + sp (for simplicity of this example)
  - thread 1 communicates s to other threads
- Orchestration
  - starting up threads
  - communicating, synchronizing with thread 1
- Mapping
  - processor j runs thread j

Administrative Issues

° **Assignment 2 will be on the home page later today**

- **Matrix Multiply contest.**
- **Find a partner (outside of your own department).**
- **Due in 2 weeks.**

° **Reading assignment**

- **www.nersc.gov/~dhbailey/cs267/Lectures/Lect04.html**
- **Optional:**
  - **Chapter 1 of Culler/Singh book**
  - **Chapters 1 and 2 of www.mcs.anl.gov/dbpp**

Cost Modeling and Performance Tradeoffs

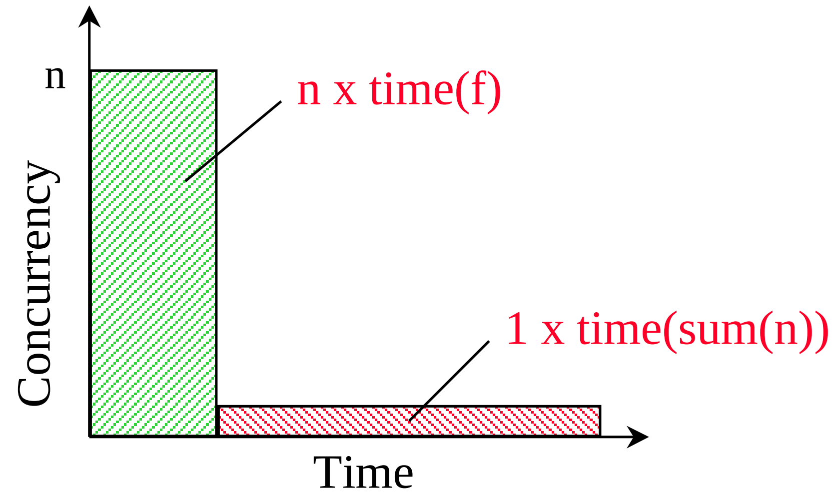
Identifying enough Concurrency

° **Parallelism profile**

- **area is total work done**

Simple Decomposition:
f (A[i]) is the parallel task

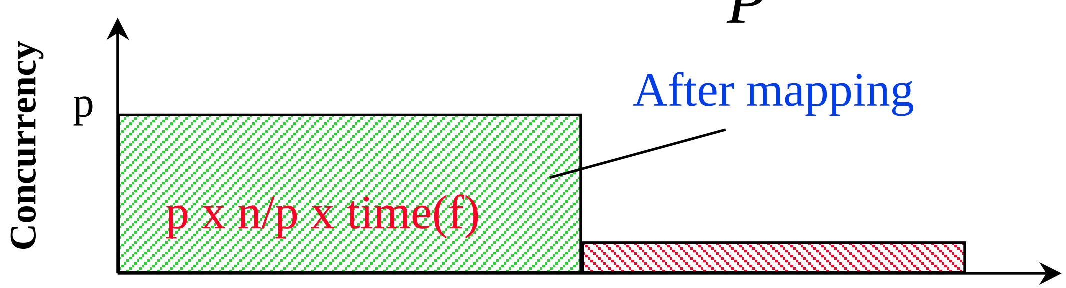
sum is sequential

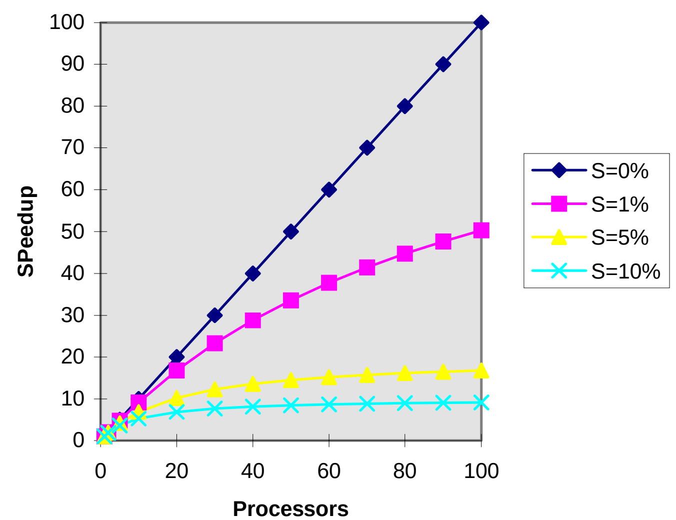


° **Amdahl's law**

- **let s be the fraction of total work done sequentially**

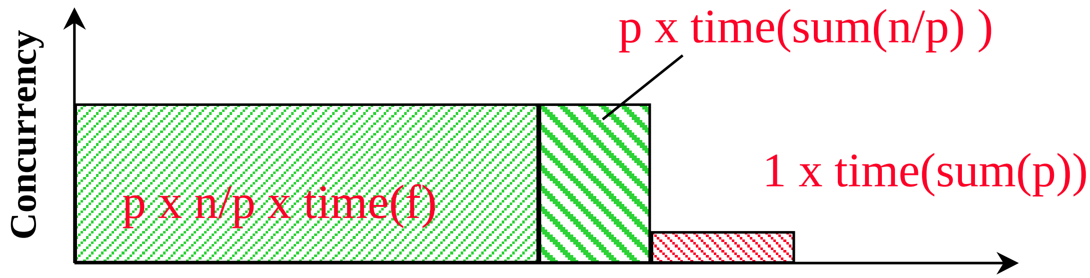
$$Speedup(P) \leq \frac{1}{s + \frac{1-s}{P}} \leq \frac{1}{s}$$





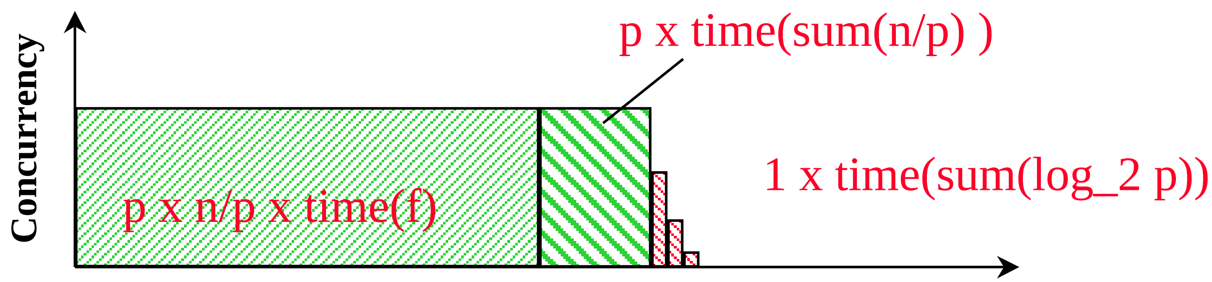
Algorithmic Trade-offs

- **Parallelize partial sum of the f's**
  - **what fraction of the computation is "sequential"**



  - **What does this do for communication? locality?**
  - **What if you sum what you "own"**

- **Parallelize the final summation (tree sum)**

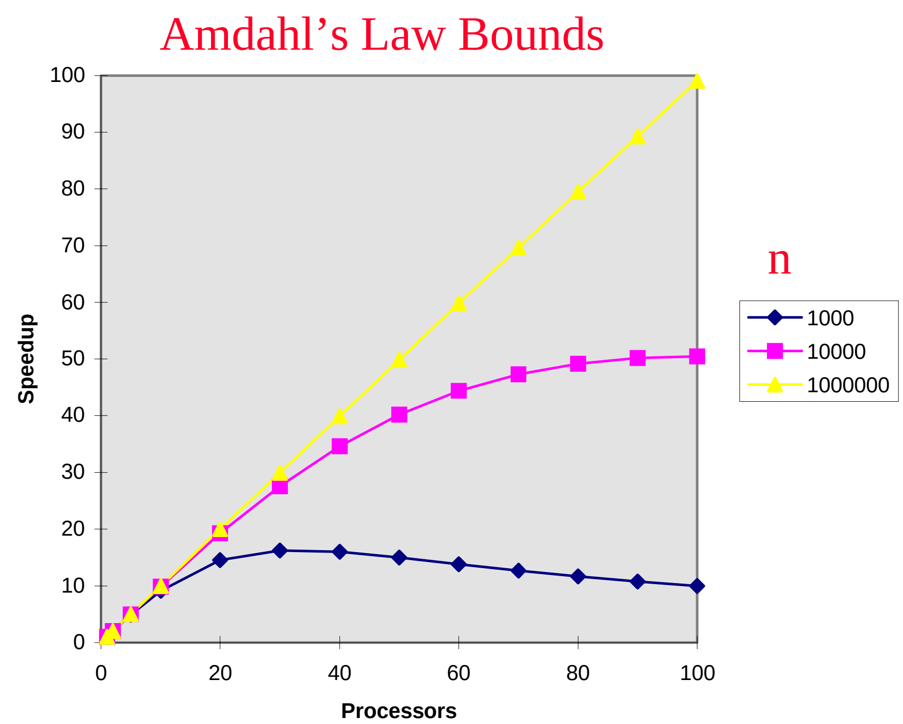


  - **Generalize Amdahl's law for arbitrary "ideal" parallelism profile**

Problem Size is Critical

- **Suppose Total work= n + P**
- **Serial work: P**
- **Parallel work: n**
- **s = serial fraction**

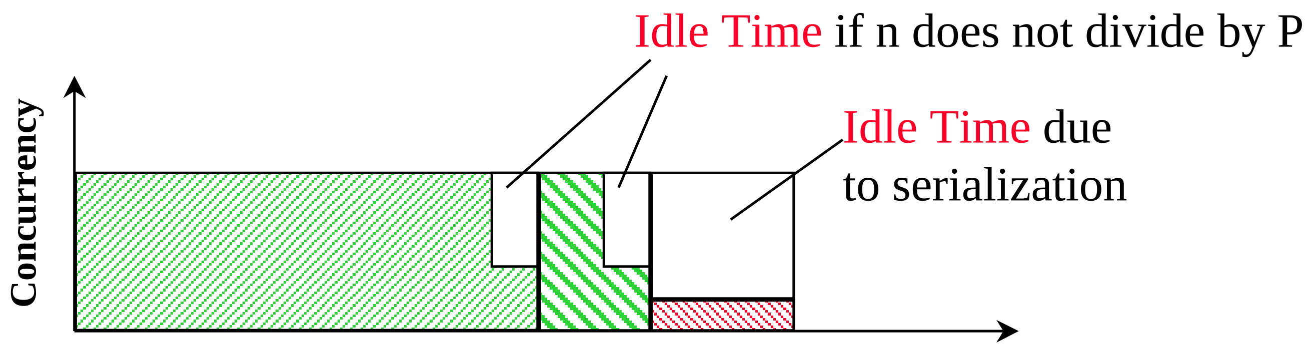
  = P/ (n+P)



In general, seek to exploit a large fraction of the peak parallelism in the problem.

Load Balancing Issues

° Insufficient concurrency will appear as load imbalance.



° Use of coarser grain tends to increase load imbalance.

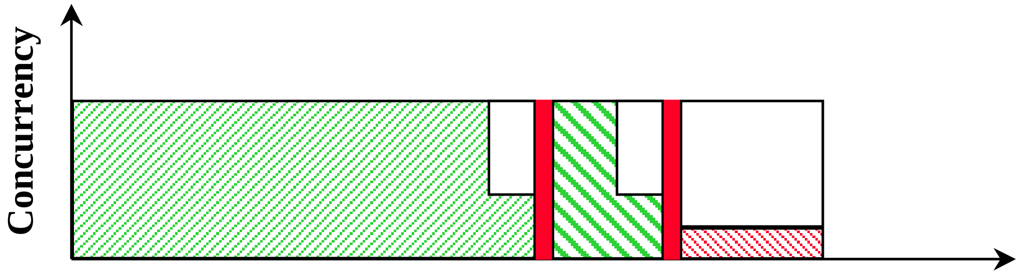
° Poor assignment of tasks can cause load imbalance.

° Synchronization waits are instantaneous load imbalance

$$Speedup(P) \leq \frac{Work(1)}{\max_p(Work(p) + idle)}$$

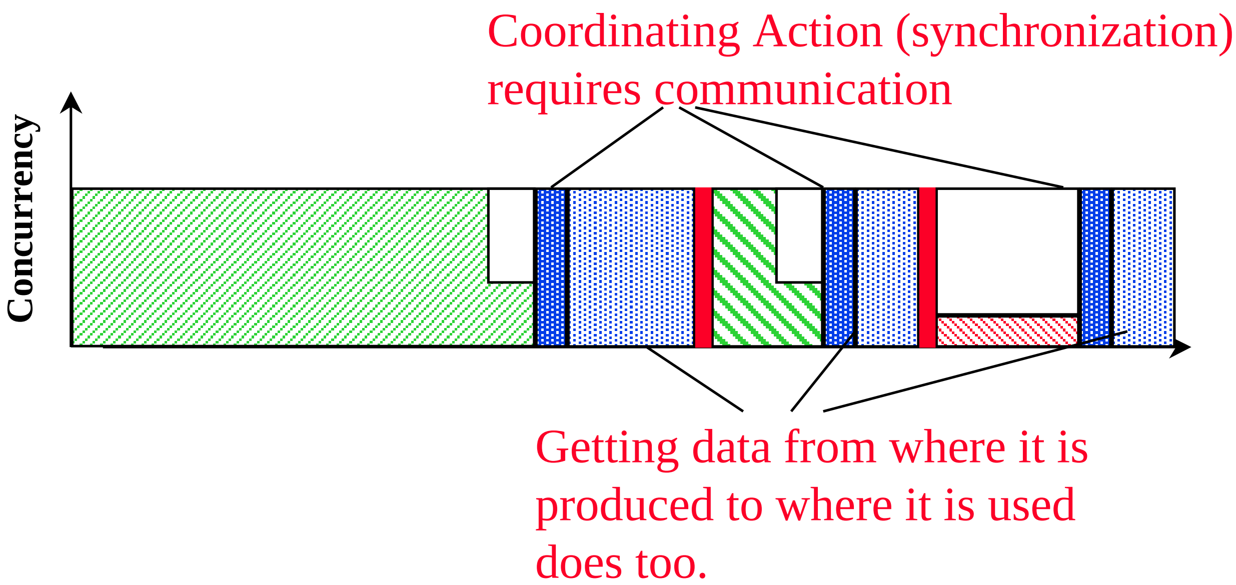
Extra Work

- There is always some amount of extra work to manage parallelism -- e.g. deciding who is to do what.



$$Speedup(P) \le \frac{Work(1)}{\mathrm{Max}_p(Work(p) + idle + extra)}$$

Communication and Synchronization

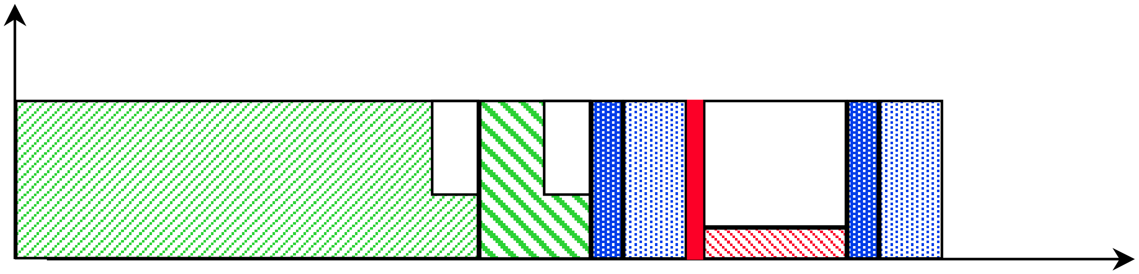


$$Speedup(P) \le \frac{Work(1)}{\max(Work(P) + idle + extra + comm)}$$

° **There are many ways to reduce communication costs.**

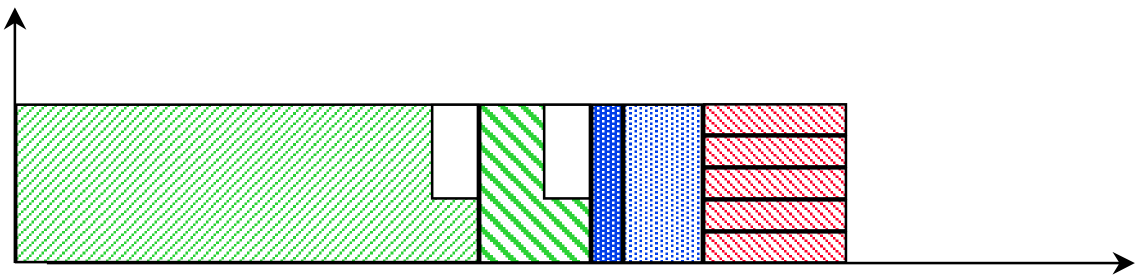
Reducing Communication Costs

° **Coordinating placement of work and data to eliminate unnecessary communication.**



° **Replicating data.**

° **Redundant work.**



° **Performing required communication efficiently.**

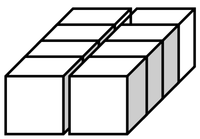
- **e.g., transfer size, contention, machine specific optimizations**

The Tension

$$Speedup(P) \le \frac{Work(1)}{\max(Work(P) + idle + comm + extraWork)}$$

Minimizing one tends to increase the others

- **Fine grain decomposition and flexible assignment tends to minimize load imbalance at the cost of increased communication**
  - **In many problems communication goes like the surface-to-volume ratio**
  - **Larger grain => larger transfers, fewer synchronization events**



- **Simple static assignment reduces extra work, but may yield load imbalance**

The Good News

- The basic work component in the parallel program may be more efficient than in the sequential case.
    - Only a small fraction of the problem fits in cache.
    - Need to chop problem up into pieces and concentrate on them to get good cache performance.
    - Similar to the parallel case.
    - Indeed, the best sequential program may emulate the parallel one.
- Communication can be hidden behind computation.
    - May lead to better algorithms for memory hierarchies.
- Parallel algorithms may lead to better serial ones.
    - Parallel search may explore space more effectively.